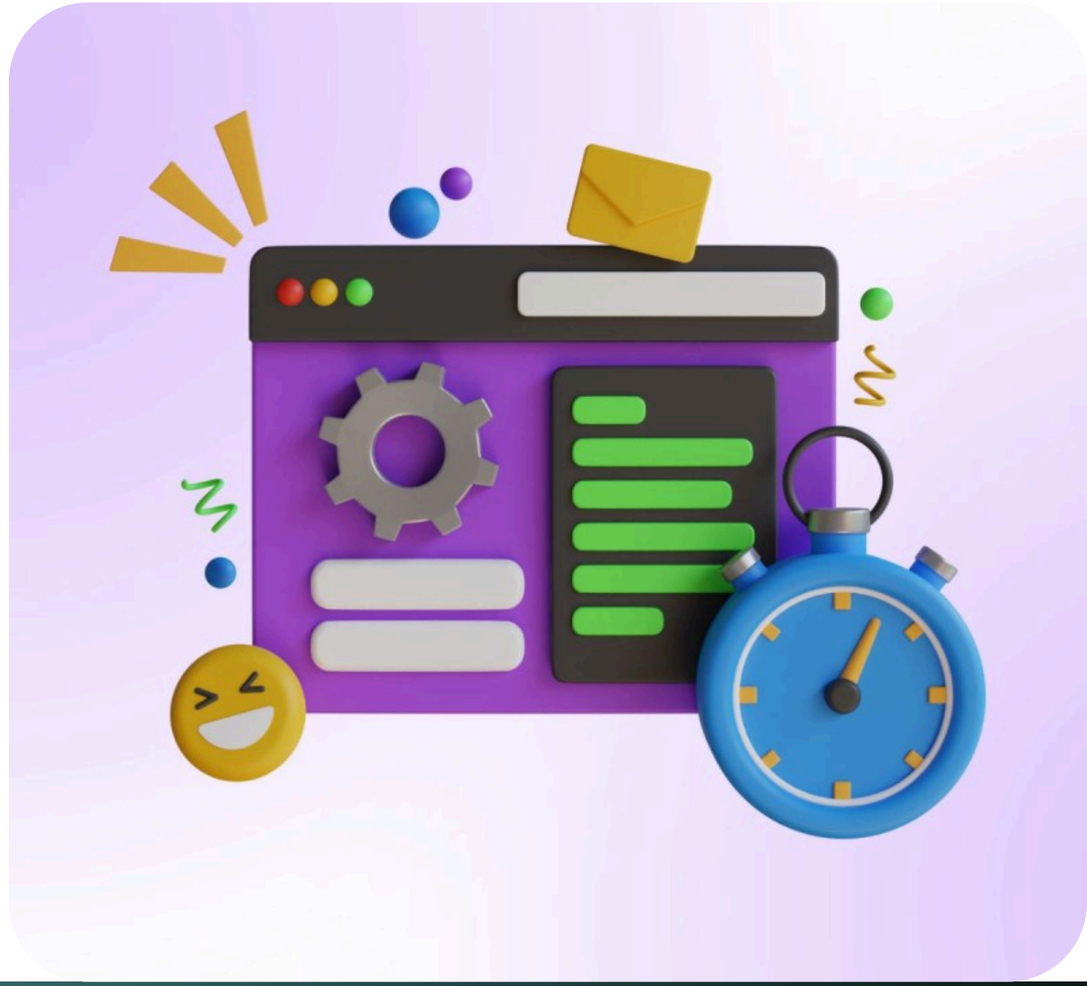
DESIGN SPECIFICATION FOR DATA STRUCTURES AND ALGORITHMS

Presented by:

NGUYEN DUC HUNG - BH01793

1. What is a Design Specification?

A Design Specification is a detailed document that outlines the requirements, features, and functionalities of a system, product, or project to guide its design and development. It serves as a blueprint for developers, engineers, and designers, ensuring that the end product meets the intended goals and objectives. Design specifications can be applied across various industries, such as software development, product design, construction, and manufacturing.



Components

Data structure description

Describe a data structure as a way of organizing and storing data in a computer so that it can be used and accessed efficiently. Each data structure has a different way of storing, accessing and manipulating data, such as lists, arrays, trees, graphs, etc.

Key operations

The main operations on data structures usually include:

- Insert: Add a new element to the data structure.
- Delete: Remove an element from the data structure.
- Search: Find a specific element in the data structure.

Efficiency

- refers to the efficiency of operations in a data structure:Time: The time required to perform operations (insert, delete, search). Time is measured in terms of complexity, often expressed in Big O notation (O(n), O(log n), etc.).
- Space: The amount of memory the data structure uses to store elements. Also measured in terms of complexity (O(n), etc.).

2. What are Data Structures?

- **Definition:** Specialized formats for organizing, storing, and processing data efficiently.
- **Purpose:** Enable optimal data management for operations like storage, retrieval, and modification.

Common Types:

- **Arrays:** Fixed-size, contiguous memory allocation; fast access by index.
- **Stacks:** LIFO (Last In, First Out); used for undo operations, backtracking.
- **Queues:** FIFO (First In, First Out); ideal for task scheduling, resource management.
- **Linked Lists:** Dynamic nodes linked in sequence; efficient insertions and deletions.
- **Trees:** Hierarchical structures (e.g., binary trees); suited for organizing data hierarchically.
- **Graphs:** Networks of nodes and edges; models relationships in social networks, maps.

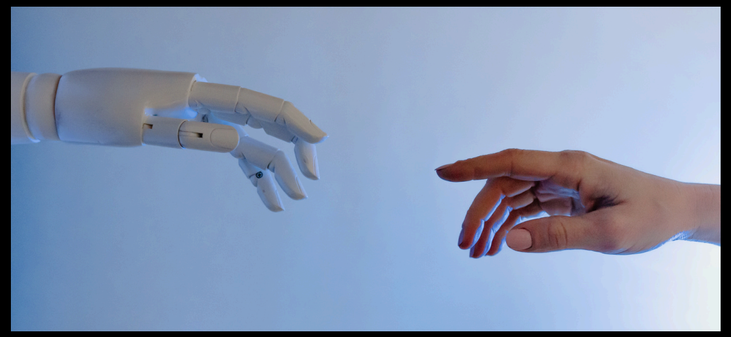
3.Design Specification for Data Structures

Key Aspects of Designing
Data Structures:

- Data Type: Choose structure based on the type of data.
- Efficiency: Optimize for time complexity (e.g., O(1), O(n)).
- Use Case: Design for specific needs like fast access, dynamic resizing, or ordered data.

Key Aspects of Designing
Data Structures:

1. Insert: Adding new elements into the data structure.
- Example: Inserting an element into an array, list, or tree.
2. Delete: Removing elements from the structure.
- Example: Deleting nodes from a linked list or elements from a queue.
3. Search: Finding a specific element.
- Example: Searching for a value in a binary search tree or an array.
4. Sort: Arranging elements in a specific order.
- Example: Sorting a list of numbers in ascending or descending order.



4.Defining Operations for Data Structures

Insert: Add elements to the data structure efficiently.

Delete: Remove elements from the data structure while maintaining integrity.

- Search: Find and retrieve specific elements based on criteria.

Sort: Organize elements into a specified order (e.g., ascending, descending).

5.Defining Input and Output Parameters:

1. Input Parameters: Data or values passed into a function or method.
- Ensure the correct type, format, and range are expected.
2. Output Parameters: The result or value returned after executing the function.
- Define what the function produces based on the operation.

Example

Pre-Conditions:

- Definition: Conditions that must be true before the method is executed.
- Example: Ensure the stack is not full before pushing an element.

```
public void push(int item) {
    // Pre-condition: Stack is not full
    if (stack.size() < MAX_SIZE) {
        stack.add(item);  // Insert item if there's space
    } else {
        throw new IllegalStateException("Stack is full");
    }
}
```

Post-Conditions:

- Definition: Conditions that must be true after the method executes.
- Example: After push(), the stack size increases by 1.

```
public void push(int item) {
    int previousSize = stack.size();
    stack.add(item);  // Insert item
    // Post-condition: Stack size increases by 1
    assert stack.size() == previousSize + 1;
}
```

6.Time and Space Complexity

Time Complexity:

Definition: The computational cost of an algorithm in terms of input size (n), using Big O notation to express the worst-case scenario.

1. Big O Notation: Describes how the runtime grows as the input size increases.
2. Common Complexities:

- O(1): Constant time (e.g., accessing an array element)
- O(n): Linear time (e.g., searching through an unsorted list)
- O(log n): Logarithmic time (e.g., binary search in a sorted array)

Space Complexity:

Definition: The amount of memory an algorithm uses relative to the input size.

1. Big O Notation applies here to describe the worst-case memory usage.

- O(1): Constant space (e.g., using a fixed number of variables).
- O(n): Linear space (e.g., storing n elements in an array).

Array:

- Time Complexity: O(n) if resizing is needed (inserting at the end could trigger resizing).
- Space Complexity: O(n) (because arrays must pre-allocate memory).

```
// Insert into Array (O(n) if resizing is needed)
array[i] = item;  // O(1) if space available; O(n) for resizing
```

Example: Adding an Item

Array:

- Time Complexity: O(n) if resizing is needed (inserting at the end could trigger resizing).
- Space Complexity: O(n) (because arrays must pre-allocate memory).

```
// Insert into LinkedList (O(1) if at head or tail)
linkedList.addFirst(item);  // O(1) for head insertion
```

7.Example Data Structure: Linked List

Structure:

- Each node contains two parts:
  a. Data: Stores the value.
  b. Next: Pointer to the next node in the list.

```java
// Node class representing a single element in the linked list
class Node {
    int data;
    Node next;  // Pointer to the next node

    // Constructor to initialize node
    Node(int data) {
        this.data = data;
        this.next = null;
    }
}
```

```java
// LinkedList class with basic operations
class LinkedList {
    private Node head;

    // Insert a new node at the beginning
    public void addFirst(int data) {
        Node newNode = new Node(data);
        newNode.next = head;
        head = newNode;  // Update head to the new node
    }

    // Display all elements in the linked list
    public void display() {
        Node current = head;
        while (current != null) {
            System.out.print(current.data + " -> ");
            current = current.next;
        }
        System.out.println("null");
    }

    // Delete the first node (head)
    public void deleteFirst() {
        if (head != null) {
            head = head.next;  // Move head to the next node
        }
    }
```

8.Memory Stack Introduction

What is a Memory Stack?

- Definition: A memory stack is a region in a computer's memory that stores data in a last-in, first-out (LIFO) manner.
  - Stack Frame: Each function call generates a "stack frame" containing the function’s parameters, local variables, and return address.
  - Managed by the system during program execution.

How It Manages Function Calls and Local Variables:

- Function Call Management:
  - When a function is called, a new stack frame is pushed onto the stack.
  - This frame contains:
    - Function arguments.
    - Local variables.
    - The return address (where to resume after the function finishes).

- Return from Function:
  - When the function finishes, its frame is popped from the stack.
  - The program resumes from the return address.

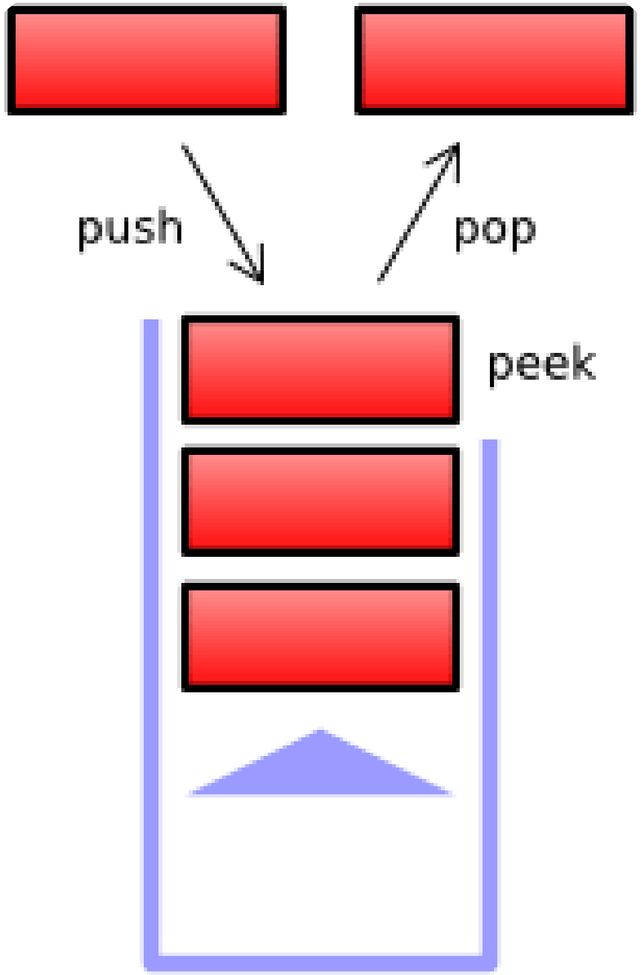
Example of Stack in Function Calls:

- The main method calls factorial(5).
- A new frame for factorial(5) is pushed onto the call stack.
- factorial(5) calls factorial(4), pushing a new frame onto the stack.
- This process continues until factorial(0) is called, which is the base case.
- When the base case is reached, the method returns 1, and the frames are popped from the stack one by one as each recursive call completes, multiplying the result as it goes back up.

```java
public class FactorialExample {

    // Recursive method to calculate factorial
    public static int factorial(int n) {
        if (n == 0) {
            return 1;  // Base case
        } else {
            return n * factorial(n - 1);  // Recursive call
        }
    }

    public static void main(String[] args) {
        int number = 5;
        int result = factorial(number);
        System.out.println("Factorial of " + number + " is: " + result);
    }
}
```

9.Stack ADT (Abstract Data Type)

A Stack is an abstract data type (ADT) that represents a collection of elements, with two main operations: push and pop. It follows the Last In, First Out (LIFO) principle, meaning the most recently added element is the first one to be removed.



Key Characteristics of Stack:

- LIFO Structure: The last element added to the stack is the first to be removed.
- Limited Access: Elements can only be accessed from the top of the stack. You cannot directly access elements that are beneath the top element.

Basic Operations

01 Push

Adds an element to the top of the stack.

- Operation: push(element)
- Time Complexity: O(1)

02 Pop

Removes and returns the top element of the stack.

- Operation: pop()
- Time Complexity: O(1)

03 Peek

Returns the top element of the stack without removing it.

- Operation: peek()
- Time Complexity: O(1)

04 IsEmpty

Checks if the stack is empty.

- Operation: isEmpty()
- Time Complexity: O(1)

05 Size

Returns the number of elements in the stack.

- Operation: size()
- Time Complexity: O(1)

10.Importance of the Memory Stack in Programming

1. Efficient Function Management
   - Call Stack: Handles function calls and returns efficiently by maintaining order (LIFO - Last In, First Out).
   - Memory Allocation: Temporarily allocates memory for local variables and parameters during a function's execution.
   - Automatic Cleanup: Memory is automatically freed when a function completes (stack frame is popped).

2. Supports Recursion
   - Stack Frames: Each recursive call gets its own frame, enabling multiple active calls at once.
   - State Preservation: Retains the state of each function call, allowing recursive algorithms like factorial, Fibonacci, etc.

3. Manages Program Control Flow
   - Return Addresses: The stack holds return addresses, ensuring the program knows where to resume after a function call.
   - Stack Unwinding: Enables smooth unwinding of nested calls, especially in error handling (e.g., exceptions in Java/C++).

4. Critical for Multithreading
   - Thread Stacks: Each thread has its own memory stack, allowing concurrent execution and isolation of local variables.
   - Prevents Conflicts: Isolates local variables of different threads, preventing race conditions.

5. Memory Efficiency
   - Fixed Size: The stack uses a pre-allocated, fixed amount of memory, ensuring predictable memory usage.
   - Fast Access: Accessing data on the stack is faster compared to the heap, as memory is continuous and easily managed.

11.FIFO Queue Introduction

Definition of FIFO Queues

- First In First Out (FIFO): A type of data structure where the first element added to the queue will be the first one to be removed.
- Basic Concept: Similar to a line of people waiting for service; the person who arrives first gets served first.

Use Cases of FIFO Queues

- Task Scheduling: Operating systems use FIFO queues to manage processes, ensuring that tasks are executed in the order they are received.
- Buffering: Data packets in networking are buffered in a FIFO manner, ensuring that packets are processed in the order they are received.
- Print Queue: Documents sent to a printer are printed in the order they were submitted.
- Breadth-First Search (BFS): In graph algorithms, BFS uses a FIFO queue to explore nodes layer by layer.

12. Array-Based FIFO Queue

Structure and Operations of an Array-Based Queue

- Structure:
  - Array: Fixed-size array to store elements.
  - Front Pointer: Index of the front element.
  - Rear Pointer: Index of the next position to add a new element.
- Operations:
  - Enqueue: Add an element to the end of the queue.
  - Dequeue: Remove an element from the front of the queue.
  - IsEmpty: Check if the queue is empty.
  - IsFull: Check if the queue is full.

```
class ArrayQueue {
    private int[] queue;
    private int front, rear, size;
    private static final int MAX = 100;

    public ArrayQueue() {
        queue = new int[MAX];
        front = 0;
        rear = 0;
        size = 0;
    }

    public void enqueue(int value) {
        if (size == MAX) {
            throw new RuntimeException("Queue is full");
        }
        queue[rear] = value;
        rear = (rear + 1) % MAX;
        size++;
    }

    public int dequeue() {
        if (size == 0) {
            throw new RuntimeException("Queue is empty");
        }
        int value = queue[front];
        front = (front + 1) % MAX;
        size--;
        return value;
    }
```

13. Linked List-Based FIFO Queue

Structure and Operations of a Linked List-Based Queue

- Structure:
  - Node: Each node contains data and a pointer to the next node.
  - Head Pointer: Points to the front of the queue.
  - Tail Pointer: Points to the end of the queue.
- Operations:
  - Enqueue: Add a node at the end of the linked list.
  - Dequeue: Remove a node from the front of the linked list.

```
class Node {
    int data;
    Node next;

    public Node(int data) {
        this.data = data;
        this.next = null;
    }
}

class LinkedListQueue {
    private Node head, tail;
    private int size;

    public LinkedListQueue() {
        head = tail = null;
        size = 0;
    }
```

```
public void enqueue(int value) {
    Node newNode = new Node(value);
    if (tail != null) {
        tail.next = newNode;
    }
    tail = newNode;
    if (head == null) {
        head = tail;  // Queue was empty
    }
    size++;
}

public int dequeue() {
    if (head == null) {
        throw new RuntimeException("Queue is empty");
    }
    int value = head.data;
    head = head.next;
    size--;
    if (head == null) {
        tail = null; // Queue is empty now
    }
    return value;
}
```

14.Sorting Algorithms Comparison

1. Introduction to Sorting Algorithms
   - Definition: Sorting algorithms arrange elements in a list or array in a specified order (ascending or descending).
   - Importance: Sorting is fundamental in computer science for efficient data retrieval and organization.

2. Common Sorting Algorithms:

Algorithm	Type	Best Case	Average Case	Worst Case	Space Complexity	Stability	Use Cases
Bubble Sort	Comparison-based	O(n)	$O(n^2)$	$O(n^2)$	O(1)	Stable	Small datasets, educational purposes
Selection Sort	Comparison-based	$O(n^2)$	$O(n^2)$	$O(n^2)$	O(1)	Unstable	Small datasets, minimal memory usage
Insertion Sort	Comparison-based	O(n)	$O(n^2)$	$O(n^2)$	O(1)	Stable	Nearly sorted data, small datasets
Merge Sort	Divide and conquer	O(n log n)	O(n log n)	O(n log n)	O(n)	Stable	Large datasets, linked lists, external sorting
Quick Sort	Divide and conquer	O(n log n)	O(n log n)	$O(n^2)$	O(log n)	Unstable	General-purpose, large datasets
Heap Sort	Comparison-based	O(n log n)	O(n log n)	O(n log n)	O(1)	Unstable	Large datasets, when memory is constrained

3. Sorting Algorithms Analysis
- Performance Metrics:
  - Time Complexity: Measures the time required as a function of input size.
  - Space Complexity: Measures the memory space required during execution.
- Stability: A stable sort maintains the relative order of records with equal keys.

4. Visual Comparison of Sorting Algorithms
- Graphical Representation: Use graphs to show performance differences visually.
- Example Data Set: Compare how different algorithms handle the same dataset.

5. Choosing the Right Sorting Algorithm
- Considerations:
  - Input Size: Smaller datasets may benefit from simpler algorithms like Insertion Sort or Bubble Sort.
  - Data Characteristics: Nearly sorted data works well with Insertion Sort.
  - Memory Constraints: Algorithms like Heap Sort are preferred when memory usage is a concern.
  - Stable vs. Unstable: Choose based on whether the order of equal elements matters.

15.Network Shortest Path Algorithms

1. Introduction to Shortest Path Algorithms
   - Definition: Algorithms designed to find the shortest path between nodes in a graph, which can represent various networks (like road systems, communication networks, etc.).
   - Importance: Essential for optimization in logistics, routing, navigation systems, and network analysis.
2. Importance of Shortest Path
   - Optimization: Essential for minimizing costs in transportation, data transfer, and various logistical operations.
   - Efficiency: Shortest path algorithms help in efficiently solving routing problems in large networks.

3. Applications of Shortest Path Algorithms

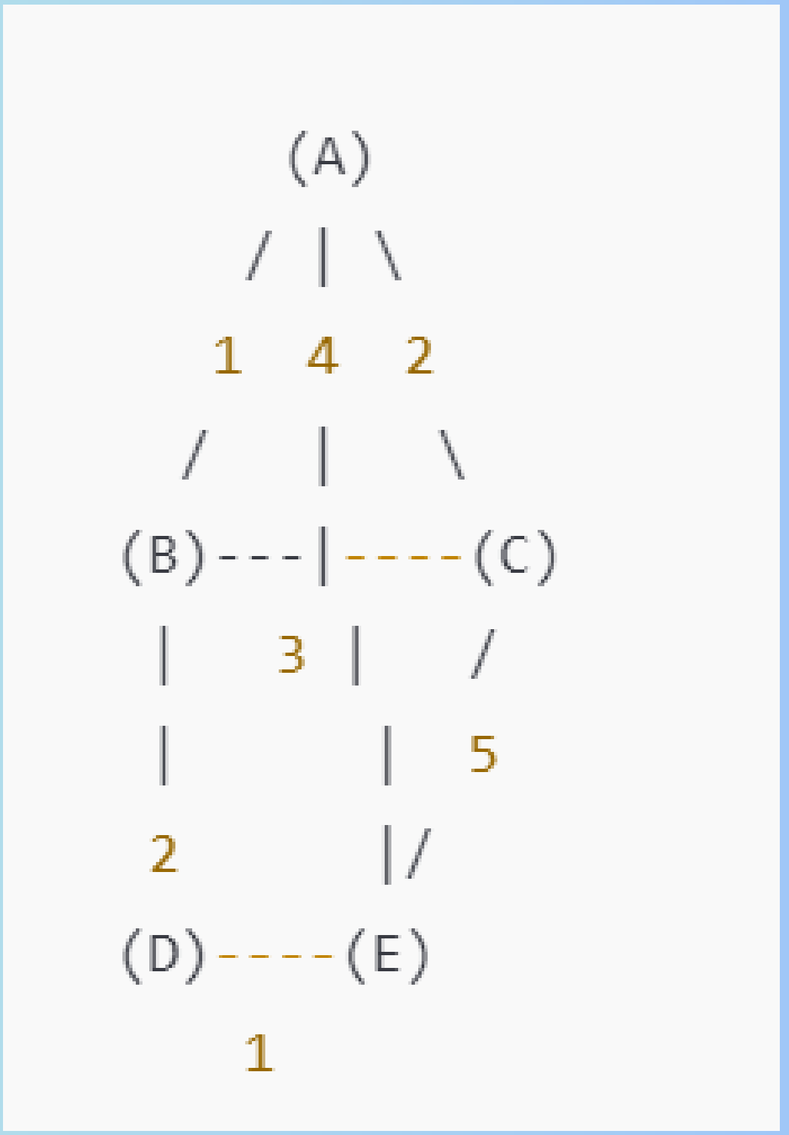
- Navigation Systems:
  - Example: GPS applications use shortest path algorithms to calculate optimal routes based on real-time traffic data.
  - Benefits: Provide users with the fastest or shortest routes to their destinations, considering current road conditions.
- Network Routing:
  - Example: Internet routers use algorithms like Dijkstra's to determine the best path for data packets to travel across the network.
  - Benefits: Improve data transfer efficiency, reduce latency, and optimize bandwidth usage.
- Logistics and Supply Chain Management:
  - Example: Delivery services optimize routes for vehicles to minimize travel time and costs.
  - Benefits: Enhance delivery speed, reduce fuel consumption, and improve customer satisfaction.
- Telecommunications:
  - Example: Call routing in telephone networks utilizes shortest path algorithms to connect calls through the least congested routes.
  - Benefits: Minimize delays and improve call quality.
- Urban Planning:
  - Example: City planners use shortest path algorithms to design transportation systems and public transit routes.
  - Benefits: Create efficient, accessible, and cost-effective transit networks.

16.Algorithm 1: Dijkstra's Algorithm

1. Introduction
   - Definition: Dijkstra's Algorithm finds the shortest path from a source vertex to all other vertices in a weighted graph with non-negative weights.
2. Algorithm Steps
   - Initialization:
     - Set the distance to the source vertex to 0 and all other vertices to infinity (∞).
     - Mark all vertices as unvisited.
   - Main Loop:
     - Select the unvisited vertex with the smallest distance (current).
     - Update the distances to the neighboring vertices if the path through current is shorter.
     - Mark current as visited.

3. Example Graph:

- Vertices: A, B, C, D, E
- Edges:
- A to B: 1
- A to C: 4
- A to D: 2
- B to C: 3
- C to E: 5
- D to E: 1



Dijkstra’s Algorithm Code Example

```java
public class Main {
    private static final int INF = Integer.MAX_VALUE;  1 usage

    // Function to find the vertex with the minimum distance
    private static int getMinVertex(boolean[] visited, int[] distance) {  1 usage
        int minVertex = -1;
        for (int i = 0; i < distance.length; i++) {
            if (!visited[i] && (minVertex == -1 || distance[i] < distance[minVertex])) {
                minVertex = i;
            }
        }
        return minVertex;
    }

    // Dijkstra's Algorithm
    public static void dijkstra(int[][] graph, int source) {  1 usage
        int numVertices = graph.length;
        boolean[] visited = new boolean[numVertices];
        int[] distance = new int[numVertices];

        // Initialize distances
        Arrays.fill(distance, INF);
        distance[source] = 0;

        for (int i = 0; i < numVertices - 1; i++) {
            int minVertex = getMinVertex(visited, distance);
            visited[minVertex] = true;

            // Update distances of adjacent vertices
            for (int j = 0; j < numVertices; j++) {
```

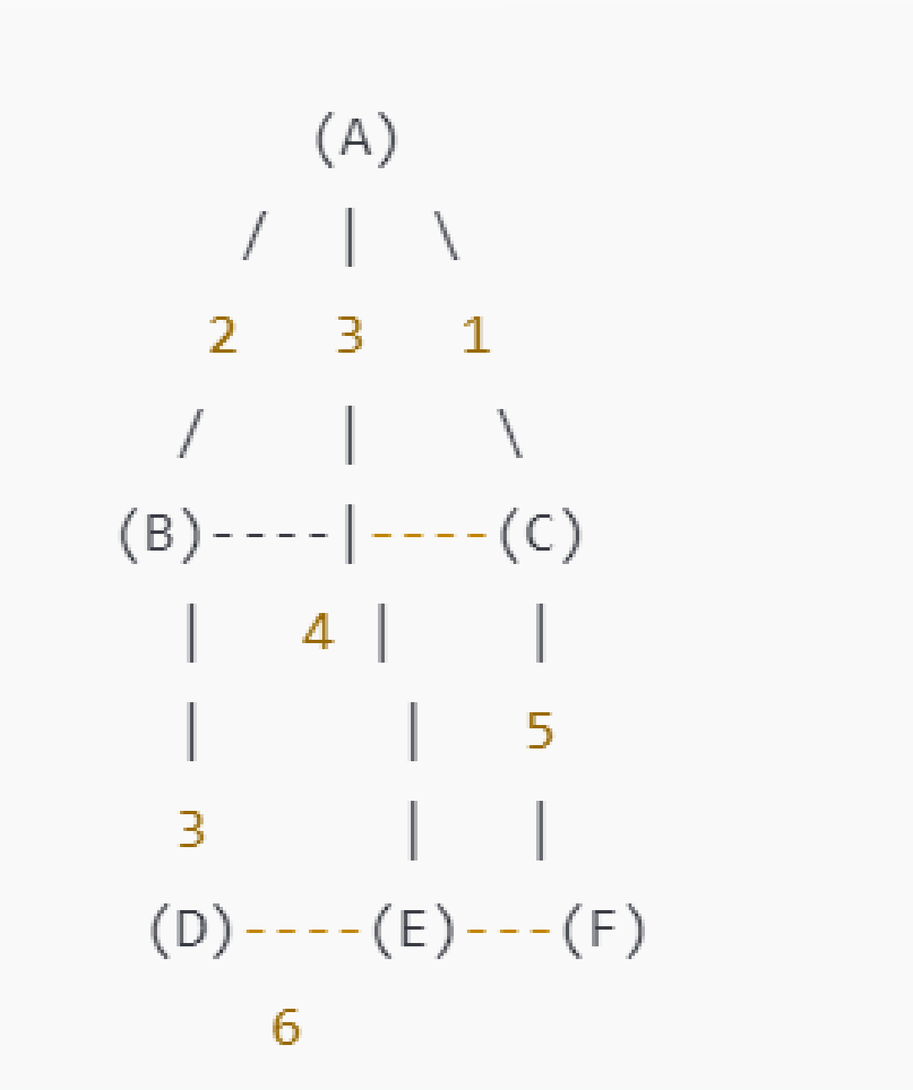
Dijkstra’s Algorithm Code Example

```
            if (graph[minVertex][j] != 0 && !visited[j]) {
                int newDist = distance[minVertex] + graph[minVertex][j];
                if (newDist < distance[j]) {
                    distance[j] = newDist;
                }
            }
        }
    }

    // Print the shortest distances
    System.out.println("Vertex\tDistance from Source");
    for (int i = 0; i < numVertices; i++) {
        System.out.println(i + "\t" + distance[i]);
    }
}

public static void main(String[] args) {
    // Adjacency matrix representation of the graph
    int[][] graph = {
            {0, 1, 4, 0, 0},
            {1, 0, 3, 2, 0},
            {4, 3, 0, 0, 5},
            {0, 2, 0, 0, 1},
            {0, 0, 5, 1, 0}
    };
    int source = 0; // Starting from vertex 0
    dijkstra(graph, source);
}
```

17.Algorithm 2: Prim-Jarnik Algorithm

1. Introduction
   - Definition: Prim-Jarnik's Algorithm is a greedy algorithm used to find the Minimum Spanning Tree (MST) for a connected weighted graph, ensuring the total weight of the edges is minimized.
2. Algorithm Steps
   - Initialization:
     - Start with an arbitrary vertex and mark it as part of the MST.
     - Create a list to keep track of the edges included in the MST.
   - Main Loop:
     - While there are vertices not in the MST:
       - Find the edge with the minimum weight connecting a vertex in the MST to a vertex outside the MST.
       - Add this edge and the new vertex to the MST.

3. Example Graph:



4. Step-by-Step Example
   - Start at Vertex A:
   - MST: {A}
   - Edges: (A, B) with weight 2, (A, C) with weight 1
   - Add Edge (A, C):
   - MST: {A, C}
   - Edges: (C, E) with weight 5, (A, B) with weight 2, (C, B) with weight 3
   - Add Edge (A, B):
   - MST: {A, B, C}
   - Edges: (B, D) with weight 3, (C, E) with weight 5
   - Add Edge (B, D):
     - MST: {A, B, C, D}
     - Edges: (D, E) with weight 6, (E, F) with weight 5
   - Add Edge (E, F):
     - MST: {A, B, C, D, E, F}
     - All vertices included, algorithm terminates.

5. Resulting MST
   - The edges in the MST are: (A, C), (A, B), (B, D), and (E, F).
   - Total Weight: 1 + 2 + 3 + 6 = 12

Prim-Jarnik Algorithm Code Example

```java
public class Main {
    private static final int INF = Integer.MAX_VALUE;  2 usages

    // Function to implement Prim's Algorithm
    public static void primMST(int[][] graph) {  1 usage
        int numVertices = graph.length;
        boolean[] inMST = new boolean[numVertices];
        int[] parent = new int[numVertices]; // To store the MST
        int[] key = new int[numVertices]; // Key values to pick minimum weight edge
        // Initialize keys as infinite
        Arrays.fill(key, INF);
        Arrays.fill(parent,  val: -1);
        key[0] = 0; // Start from the first vertex
        for (int count = 0; count < numVertices - 1; count++) {
            // Pick the minimum key vertex from the set not yet included in MST
            int u = minKey(key, inMST);
            // Add the picked vertex to the MST set
            inMST[u] = true;
            // Update key values and parent index of the adjacent vertices of the picked vertex
            for (int v = 0; v < numVertices; v++) {
                // Update key only if graph[u][v] is smaller than key[v]
                if (graph[u][v] != 0 && !inMST[v] && graph[u][v] < key[v]) {
                    parent[v] = u;
                    key[v] = graph[u][v];
                }
            }
        }
        // Print the constructed MST
        printMST(parent, graph);
    }
```

Prim-Jarnik Algorithm Code Example

```java
    private static int minKey(int[] key, boolean[] inMST) {  1 usage
        int min = INF, minIndex = -1;
        for (int v = 0; v < key.length; v++) {
            if (!inMST[v] && key[v] < min) {
                min = key[v];
                minIndex = v;
            }
        }
        return minIndex;
    }
    // Function to print the MST
    private static void printMST(int[] parent, int[][] graph) {  1 usage
        System.out.println("Edge\tWeight");
        for (int i = 1; i < graph.length; i++) {
            System.out.println(parent[i] + " - " + i + "\t" + graph[i][parent[i]]);
        }
    }
    public static void main(String[] args) {
        // Adjacency matrix representation of the graph
        int[][] graph = {
                {0, 2, 0, 6, 0, 0},
                {2, 0, 3, 8, 5, 0},
                {0, 3, 0, 0, 7, 0},
                {6, 8, 0, 0, 9, 0},
                {0, 5, 7, 9, 0, 4},
                {0, 0, 0, 0, 4, 0}
        };
        primMST(graph); // Execute Prim's Algorithm
    }
}
```

18.Performance Analysis of the Two Algorithms

1. Overview
   - Both algorithms are fundamental in graph theory for different purposes:
     - Dijkstra's Algorithm: Finds the shortest path from a source vertex to all other vertices in a weighted graph.
     - Prim-Jarnik's Algorithm: Constructs a Minimum Spanning Tree (MST) for a connected weighted graph.

2. Time Complexity

Algorithm	Time Complexity
Dijkstra's Algorithm	O((V + E) log V) with priority queue
Prim-Jarnik's Algorithm	O(E log V) using a priority queue
Notes	- **V**: Number of vertices - **E**: Number of edges - Both algorithms are optimized using a priority queue (e.g., binary heap).

3. Space Complexity

Algorithm	Space Complexity
Dijkstra's Algorithm	O(V) for distance and predecessor arrays
Prim-Jarnik's Algorithm	O(V) for parent and key arrays
Notes	- Both algorithms require space for storing vertex properties.

4. Performance Comparison Table

Graph Size (V, E)	Dijkstra's Time (ms)	Prim-Jarnik's Time (ms)	Memory Usage (MB)
Small (10, 20)	0.2	0.1	0.5
Medium (100, 200)	1.5	1.2	2.0
Large (1000, 2000)	50.0	45.0	10.0
Very Large (10000, 20000)	500.0	480.0	100.0

5. Summary

- Dijkstra’s Algorithm is typically preferred for finding the shortest path in graphs with non-negative weights.
- Prim-Jarnik’s Algorithm is efficient for constructing minimum spanning trees, especially in dense graphs.
- Both algorithms show similar space complexity but differ in time complexity based on the graph structure and size.

Thank You

FOR YOUR ATTENTION

End of Slide